VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Absolvování individuální odborné praxe

# Individual Professional Practice in the Company

2019 Michal Hodaň

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Zadání bakalářské práce

Student: **Michal Hodaň**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Absolvování individuální odborné praxe
Individual Professional Practice in the Company

Jazyk vypracování: čeština

Zásady pro vypracování:

1. Student vykoná individuální praxi ve firmě: MonkeyData, s.r.o.
2. Struktura závěrečné zprávy:
   a) Popis odborného zaměření firmy, u které student vykonal odbornou praxi a popis pracovního zařazení studenta.
   b) Seznam úkolů zadaných studentovi v průběhu odborné praxe s vyjádřením jejich časové náročnosti.
   c) Zvolený postup řešení zadaných úkolů.
   d) Teoretické a praktické znalosti a dovednosti získané v průběhu studia uplatněné studentem v průběhu odborné praxe.
   e) Znalosti či dovednosti scházející studentovi v průběhu odborné praxe.
   f) Dosažené výsledky v průběhu odborné praxe a její celkové zhodnocení.

Seznam doporučené odborné literatury:

Podle pokynů konzultanta, který vede odbornou praxi studenta.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **doc. Mgr. Jiří Dvorský, Ph.D.**

Konzultant bakalářské práce: Bc. Tomáš Widlák

Datum zadání: 01.09.2018
Datum odevzdání: 30.04.2019


doc. Ing. Jan Platoš, Ph.D.
*vedoucí katedry*

prof. Ing. Pavel Brandštetter, CSc.
*děkan fakulty*

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 12. dubna 2019 .....

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.

V Ostravě 12. dubna 2019

MONKEYDATA
MonkeyData s.r.o. -4-
Hladnovská 1255/23, 710 00 Ostrava
IČ: 02731452 DIČ: CZ02731452
www.monkeydata.cz

ING. JAN LASTŮVKA

Tímto bych rád poděkoval společnosti MonkeyData s.r.o. za možnost vykonání odborné praxe právě u nich, jakožto i začlenění do pracovního týmu. Dále bych chtěl poděkovat Bc. Tomáši Widlákovi za cenné rady, trpělivost a také zkušenosti o které se semnou po dobu praxe dělil. Poděkování patří i doc. Mgr. Jiřímu Dvorskému, Ph.D. za vedení a pomoc při zpracovávání této práce.

**Abstrakt**

Tato bakalářská práce popisuje činnost, vykonanou během odborné praxe ve společnosti MonkeyData s.r.o., která se zabývá vývojem vývojem webové aplikace pro analýzu dat z oblasti ecommerce podnikání. Hlavní náplní praxe bylo vyvinout systém událostí, který by zefektivnil a zjednodušil komunikaci mezi jednotlivými částmi aplikace. Součástí tohoto úkolu bylo i navržení struktury pro ukládání událostí, jakožto i zvolení technologií k tomu určených. V závěru práce shrnuji nově nabyté, chybějící i použité znalosti, včetně hodnocení celkového průběhu praxe.

**Klíčová slova**: MonkeyData s.r.o., bakalářská praxe, PHP, Laravel, MySQL, RabbitMQ, PHPUnit, Mikroservisní architektura.

**Abstract**

This thesis describes the work performed during the professional practice at MonkeyData s.r.o., a web application developer for analysis of ecommerce business data. The main goal has been to develop a event system, which would improve efficiency and further simplify communication between application parts. This task included designing a event storage structure, as well as selection of suitable technology. In conclusion I summarize newly gained, missing and used knowledge, including evaluation of practice as a whole.

**Key Words**: MonkeyData s.r.o., bachelor practice, PHP, Laravel, MySQL, RabbitMQ, PHPUnit, Microservices architecture.

# Obsah

# Seznam použitých zkratek a symbolů

| | | |
|---|---|---|
| AMQP | – | Advanced Message Queuing Protocol |
| API | – | Application Programming Interface |
| CLI | – | Command-line Interface |
| Composer | – | Balíčkovací systém pro PHP |
| Event | – | Událost |
| ETL | – | Extract, Transform, Load |
| JSON | – | Javascript Object Notation |
| Laravel | – | PHP Framework |
| Mock | – | Dublér, který kontrolovaně mimikuje chování reálného objektu |
| MySQL | – | Relační databáze založena na SQL |
| OOP | – | Object Oriented Programming |
| PHP | – | Hypertext Preprocessor, general-purpose language [1] |
| PHPUnit | – | Framework pro psaní jednotkových testů v PHP |
| POC | – | Proof of Concept |
| RabbitMQ | – | Aplikace pro zprostředkovávání zpráv, podporující AMQP |
| SQL | – | Structured Query Language |
| TDD | – | Test Driven Development |
| XML | – | Extensible Markup Language |

# Seznam obrázků

# Seznam výpisů zdrojového kódu

# 1 Úvod

Tato bakalářská práce se zabývá průběhem a náplní mé odborné praxe vykonané během bakalářského studia ve firmě MonkeyData s.r.o.. Pro tuto možnost praxe jsem se rozhodl hlavě z důvodu získání praktických zkušeností s profesionálním vývojem software, postupy a technologie, které jsou v praxi uplatněny a doplnit nebo rozšířit své teoretické znalosti nabyté při studiu. Tato práce se poté zabývá představením firmy MonkeyData s.r.o., pozicí kterou jsem ve firmě vykonával i firemním prostředím. Dále je popsána aplikace MonkeyData, na jejímž vývoji jsem se během praxe podílel, včetně základního technologického členění aplikace a jak je v praxi uživateli využívána. Poté následuje už popis úkolů, včetně problémů, které během vývoje nastaly a jejich následné řešení a implementace. V závěru se nachází zhodnocení průběhu praxe, sumarizace nabytých zkušeností a znalostí získaných během vykonávání praxe, jakožto i ohlédnutí za znalostmi získaných během studia.

# 2 Firma MonkeyData s.r.o.

Společnost MonkeyData s.r.o. se zabývá především vývojem a provozem aplikace MonkeyData, která má za cíl, analýzu dat z ecommerce a přinést prezentaci takto analyzovaných dat uživatelům ve srozumitelné a jednoduše pochopitelné formě přímo integrované do jejich eshopových řešení. Dále seznamuje a edukuje širokou veřejnost o výhodách analýz týkajících se ecommerce a bezpečnosti ukládání takovýchto dat. Vedení, vývojářský tým i marketing sídlí v kancelářích podnikatelském a inovačním centru Ostrava. Zástupci společnosti se zúčastnili mnoha tuzemských i zahraničních konferencí zaměřené na startupové firmy, nebo konference zaměřujících se na ecommerce, analýzu dat.


Obrázek 1: Logo MonkeyData

## 2.1 Pracovní zařazení

Ve firmě jsem se ucházel o pracovní pozici, kterou jsem si vybral v informačním systému KatIS. Proběhlo krátké přijímací řízení, kde jsem se poprvé setkal se svým budoucím mentorem Bc. Tomášem Widlákem se kterým proběhla diskuze na téma vývoje software, zkušeností s PHP a testováním software, také došlo na krátké otestování mých znalostí. Přijímacím řízením jsem nicméně prošel a mohl se tedy těšit na zařazení do vývojářského týmu na pozici PHP developer junior. Ihned po mém nástupu, jsem byl začleněn do šesti členného vývojářského týmu, byl mi projektový manažerem vytvořen firemní email a přidány přístupy do systémů které firma používá ke komunikaci, organizaci a správě vývojářských nástrojů.

## 2.2 Firemní prostředí

Po dobu praxe jsem spolupracoval především s kolegy z vývojářského týmu, kteří mi zpočátku i zadávali úkoly. Později po mém zapracování jsem začal také přímo spolupracovat s projektovým manažerem. Úkoly mi byly přidělovány přes aplikaci JIRA, jenž slouží k řízení projektů, které jsou rozděleny na základě mnoha rozličných atributů. Komunikace s členy týmu probíhala pomocí aplikace Slack, nebo přímým osobním kontaktem, který byl v mnoha prospěšný pro firemní kolektiv. Odvedenou práci jsem vždy nahrával do verzovacího nástroje GIT, který v mnohém usnadňuje práci více vývojářů nad stejným projektem.

# 3 Projekt MonkeyData

Společnost MonkeyData s.r.o. vyvíjí aplikaci pro e-shopy, které poskytují svá data o podnikání a chtějí takto analyzována data zobrazit srozumitelně a v moderní podobě. Jak již bylo zmíněno, aplikace zakládá na myšlence, že stáhnutá data z různých analytických nástrojů, se zpracují a zobrazí v přehledných grafech a tabulkách, které usnadňují jejich vlastníkům analýzu jak se danému projektu daří, nebo jak efektivně se využívají vynaložené finance například na reklamu.

Aplikace nabízí provozovatelům e-shopů, spoustu přehledných grafů na základě dat, jimi poskytnutých o jejich podnikání a má sloužit jako podklad pro další vývoj daného e-shopu. Grafy jsou děleny podle jejich zdroje dat a zaměření jako PPC, kde jsou zobrazeny data z internetové reklamy, pro kterou se v tuzemsku využívají především služby jako Sklik nebo Adwords. Navíc všechny tyto přehledy s grafy si může uživatel stáhnout v podobě PDF, nebo jako balíček obrázků ve formátu PNG. Aplikace také nabízí si nechat vybrané přehledy posílat ve zvolených intervalech na email tak, aby mohl být uživatel neustále informován o svém podnikání.

Z technologického hlediska je aplikace rozdělena do několika částí, které dohromady tvoří aplikaci, která zajišťuje vysokou míru dostupnosti, škálovatelnosti a výkonnosti. Celá aplikace je navíc umístěna v cloudu, poskytovaném společností Google, což zajišťuje ještě větší míru dostupnosti, protože pokud jeden ze serverů, na kterém část aplikace běží náhodou vypadne, ihned jej nahradí jiný, aniž by to uživatel jakýmkoliv způsobem poznal.

## 3.1 Import dat

První částí aplikace je importní systém, který je pro samotné uživatele skryt, ale plní jednu z nejdůležitějších funkcí a to je neustále stahování dat ze zdrojů, které si uživatel v aplikaci napojil. Celá tato část běží na oddělených serverech tak, aby nijak nenarušila běh samotné aplikace. Importní část po stažení dat vše uloží tak, aby další části aplikace mohly tyto data efektivně zpracovat a upravit do finální podoby. Důležitou informací je to, že import dat probíhá nepřetržitě tak, aby uživatelé měli v aplikaci pořád o nejaktuálnější informace.

## 3.2 Kalkulace

Kalkulace následuje po importní části. Má za úkol očistit data o přebytečné informace a normalizovat data pomocí tzv. ETL nástrojů. Ty slouží k tomu, aby finální data byla v jednotné podobě uložena do databáze a mohly být nad nimi prováděny běžné databázové operace. Z důvodů obrovského množství dat, které nejsou jen paměťové náročné, ale i výpočty nad nimi jsou procesorově náročné. Výsledná data jsou již ukládána do relační databáze a to konkrétně jako analytická databáze, kde jsou již efektivně uložena a připravena pro výběr dat do grafů.

## 3.3 Grafová část

Tato část aplikace funguje v podstatě jako API, jen je zabezpečena tak, že k ní lze přistoupit pouze z aplikace samotné. Grafová část má za úkol výběr dat z databáze a všechna tato data připravit tak, aby byla připravená na vložení do grafů. V této části je kladen důraz na výkonnost celého běhu, jelikož jsou grafy načítány přímo na výstup k uživateli, tudíž musí být získání dat pro graf v řádech maximálně stovek milisekund. Za součást grafové vrstvy se také počítá javascriptová část na straně klienta, která má za úkol vykreslit samotné grafy na výstupu s využitím knihovny dxChart [7]. Tato součást aplikace je také odpovědná za zajištění komunikace se serverem a volání příslušných API pro grafy. Také slouží k řízení obsahů grafu na základě uživatelských dat.

## 3.4 API

API slouží k získání dat z aplikace zabezpečenou formou tak, aby mohla být data využita například v mobilní aplikaci. Celé API je koncipováno formou REST API a přístup k němu je zabezpečen standardem OAuth2 [8].

## 3.5 Aplikační část

Hlavní část aplikace, se kterou se uživatel setká. Slouží k zajištění komunikace s uživatelem. Skrze tuto část si uživatel vytváří projekty, ke kterým si napojí zdroje dat, které v daném projektu využívá a chce z nich vytvořit grafy. Napojení zdroje je nejčastěji automatické nebo jen vyplnění nutných údajů k tomu, aby mohlo začít stahování dat. Také mu umožňuje sdílet své projekty s jinými uživateli, vytvářet si přehledné reporty a ty si nechat zasílat na email.

## 3.6 Mobilní Aplikace

Mobilní aplikace slouží k tomu, aby uživatel měl skrze ní neustálý přístup k datům bez nutnosti toho, aby byl nucen využít počítač. Aplikace slouží hlavně k zobrazení grafů a metrik tak jako ve webové aplikaci, nikoli však k nastavení projektů či změnu napojení zdrojů, čímž je zajištěno co nejmenší využití prostředků mobilního telefonu, nebo tabletu.

## 3.7 Event Systém

Systém událostí je kompletně nová část ekosystému, kterou jsem měl za úkol implementovat a od níž se očekává, že bude poskytovat komunikační most mezi jednotlivými částmi a zjednoduší tak nároky na správu, sníží komplexitu komunikace a zároveň tak zrychlí celkový chod celého systému.

# 4 Řešené úkoly

## 4.1 Úkol první – Implementace struktur událostí a odpovědí

### 4.1.1 Zadání úkolu

První úkolem bylo navrhnout a implementovat struktury pro událost a odpověď. Dále vybrat formát pro jejich serializaci.

### 4.1.2 Analýza úkolu

Před samotnou implementací bylo nutné definovat atributy struktur událostí a odpovědí. Zjistit, které atributy mají společné a navrhnout podle toho patřičnou hierarchii tříd, kterou posléze implementovat do nově vytvořeného Composer [4] balíčku s názvem Event a pečlivě zvážit jaký formát serializace zvolit.

### 4.1.3 Implementace

První výzva byla tedy definovat atributy těchto struktur, ty přímo vychází z použití a sice obě obsahují datum vzniku ve formátu unix timestamp a obě struktury jsou serializovatelné. Zbylá data jsou již specifická na úrovni konkrétní odpovědi, nebo události, popřípadě speciální případ struktury události, která požaduje odpověď, je dále rozšířená o objekt reprezentující odkaz na odpověď. Před implementací, bylo potřeba vytvořit zmíněný balíček, který bude obsahovat implementaci struktur událostí. Pro vytváření nových balíčků již je ve firmě nástroj integrovaný v automatizačním serveru Jenkins. Do nástroje je nutno zadat identifikační údaje o balíčku, které jsou použity pro přípravu GIT repositáře, obsahujícího základní kostru balíčku spolu s výchozím nastavením pro PHP. Do tohoto repositáře, je poté nahráván postupný vývoj balíčku. Struktury událostí a odpovědí jsou v jazyce PHP navrženy jako abstraktní třídy, vycházející ze společného předka, abstraktní třídy `APayload` obsahující celočíselný atribut, odpovídající právě datovému typu unix timestamp, který musí být předán v konstruktoru a implementující rozhraní `ISerializable`, které představuje návrhový vzor Visitor a vystavuje třídu námi definovanému druhu serializace. Rozhraní je je implementováno v neveřejném balíčku Serializer, který tento balíček používá.

```php
abstract class APayload implements ISerializable {
   /**
    * @var int
    */
   private $createdAt;

   public function __construct(int $createdAt) {
      $this->createdAt = $createdAt;
   }

   public function getCreatedAt(): int {
      return $this->createdAt;
   }
}
```

Výpis 1: Abstraktní třída APayload

Touto implementací je zaručeno, že každá konkrétní událost, nebo odpověď bude obsahovat námi požadovanou časovou značku s možností rozšíření o své specifické atributy a druhy serializace.

```php
abstract class AEvent extends APayload {
}

abstract class AEventResponse extends APayload {
}
```

Výpis 2: Abstraktní třídy AEvent a AEventResponse

Pro událost vyžadující odpověď bylo nutné vytvořit strukturu odkazu na odpověď, která bude sloužit ke spárování události a odpovědi. Tudíž musí obsahovat kód podle kterého se bude párovat a očekávanou třídu, kterou bude odpověď implementovat, aby na straně vyvolání události, byla dostupná podpora očekávaného datového typu.

```php
final class EventResponseLink {

   private $responseCode;

   private $responseClass;

   public function __construct(string $responseCode, string $responseClass) {
      $this->responseCode = $responseCode;
      $this->responseClass = $responseClass;
   }

   public function getResponseCode(): string {
      return $this->responseCode;
   }

      public function getResponseClass(): string {
      return $this->responseClass;
   }
}
```

Výpis 3: Struktura odkazu na odpověď

`ARespondableEvent` slouží jako základ pro implementaci události vyžadující odpověď a tudíž vyžaduje předání výše popsané struktury `EventResponseLink` už v konstruktoru.

```php
abstract class ARespondableEvent extends AEvent {

   private $link;

   public function __construct(EventResponseLink $link, int $createdAt) {
      parent::__construct($createdAt);
      $this->link = $link;
   }

   public function getEventResponseLink(): EventResponseLink {
      return $this->link;
   }
}
```

Výpis 4: Abstraktní třída ARespondableEvent

K dokončení prvního úkolu je tedy nutné vybrat mezi navrhovanými formáty serializace XML, binární formát a JSON, ten jenž se nejvíce hodí pro případ použití serializace dat v systému událostí. Po zvážení alternativ byl zvolen JSON pro jeho vlastnosti jako dobrá lidská čitelnost, malé nároky na režii serializovaných dat, podpora datových typů, nezávislost na platformě a všeobecně dobrou podporu v programovacích jazycích.

## 4.2 Úkol druhý – Implementace Event Queue

### 4.2.1 Zadání úkolu

Druhým úkolem bylo vytvořit Event Queue, subsystém, který bude zprostředkovávat vkládání a vybírání událostí. Tyto základní funkcionality budou nadále rozšířeny o možnost odpovědi na událost a dotazování se na událost, z daného podprostoru jmen událostí. To vše nezávislé na konkrétní technologii fronty.

### 4.2.2 Analýza úkolu

Před implementací bylo nutné navrhnout strukturu metadat pro událost, odpověď a vytvořit balíček Event Queue, který bude obsahovat samotnou implementaci. Dále definovat rozhraní, který bude splňovat všechny požadavky front a otevře jednoduché rozhraní pro implementace stojící na konkrétní technologii fronty.

### 4.2.3 Implementace

Nejdříve jsem začal vytvořením balíčku Event Queue, do kterého budou všechna definovaná rozhraní implementována a který je závislý na balíčku Event, protože používá struktury a rozhraní v něm definované. Díky již předem zmíněnému firemnímu nástroji na vytváření balíčků, bylo vytvoření, včetně závislostí hotové v rámci sekund a já jsem mohl přejít k návrhům struktur a rozhraní. V nejjednodušší verzi fronty bez rozšířené funkcionality se jedná pouze o dvě metody pro vložení a výběr události, z toho vložení přijímá jediný povinný parametr datového typu `AEvent` a návratovým typem void. Výběr zase nepřijímá žádný parametr a vrací hodnotu typu `AEvent` nebo null. V PHP takováto funkcionalita je stejně jednoduše implementována pomocí rozhraní.

```php
interface IEventQueue {
   public function enqueue(AEvent $event): void;

   public function dequeue(): ?AEvent;
}
```

Výpis 5: Rozhraní IEventQueue

Požadované rozšíření v podobě možnosti dotazování se na událost z konkrétního podprostoru jmen událostí, je nutné stávající rozhraní rozšířit o metodu přijímající řetězec reprezentující prostor jmen a vracející hodnotu `AEvent` nebo null. V PHP je takováto implementace realizována jako samostatné rozhraní, které obsahuje požadovanou metodu. Třída implementující konkrétní technologii fronty poté může, nebo nemusí implementovat toto rozhraní podle možnosti technologie, nebo případu užití.

```
interface IEventQueueQueryable {
   public function queryQueue(string $namespace): ?AEvent;
}
```

Výpis 6: Rozhraní IEventQueueQueryable

Další rozšíření, jehož rozhraní je nutné definovat, je funkcionalita odpovědi na událost, jenž vyžaduje odpověď. Jedná se o nejsložitější funkcionalitu, což se ukázalo už při definování tohoto rozhraní. Kompletní funkcionalita zaslání události, vyžadující odpověď a samotné odpovědi na událost je definována jako tři metody. První metoda reprezentující zaslání události očekávající odpověď, přijímá parametr typu `ARespondableEvent` s návratovým typem `EventResponseLink`, který sice je obsažen už v předávané události, ale pro lepší řetězení příkazů je i vrácen. Druhá metoda pro funkcionalitu samotné odpovědi na událost, přijímající argument typu `AEventResponse` a strukturu odkazu na odpověď s návratovým typem void. Poslední metoda pro vyzvednutí odpovědi z fronty, přijímá strukturu reprezentující odkaz na odpověď s návratovým typem `AEventResponse` nebo null. Funkcionality odpovědi na událost jsem se rozhodl rozdělit na dvě samostatná rozhraní. První `IEventQueueConversable` obsahující dvě metody pro zaslání události vyžadující odpověď a získání takto na párované odpovědi.

```
interface IEventQueueConversable {

   public function request(AEventRespondable $event): EventResponseLink;

   public function retrieveResponse(EventResponseLink $link): ?AEventResponse;
}
```

Výpis 7: Rozhraní IEventQueueConversable

Druhé `IEventQueueRespondable` obsahující pouze metodu pro zaslání odpovědi.

```
interface IEventQueueRespondable {

   public function respond(AEventResponse $response, EventResponseLink $link);
}
```

Výpis 8: Rozhraní IEventQueueRespondable

Tato rozdělení zaručuje větší flexibilitu při rozšiřování a také dovolují precizněji rozdělit role tříd implementujících konkrétní rozhraní a funkcí.

Po dokončení definic rozhraní jsem mohl přejít k implementaci Event Queue nezávislé na technologii fronty, navržené tak, aby pro implementaci s konkrétní technologií zbyly pouze na technologii závislé abstraktní metody, která tato abstrakce vystaví. Úkol se to nezdál jednoduchý, však předchozí návrhy rozhraní rozdělené na konkrétní role, návrh značně zjednodušily. Začal jsem tedy vytvořením abstraktní třídy `AEventQueue` implementující pouze rozhraní `IEventQueue` a pustil se do návrhu struktury sloužící jako hlavička metadat pro vložení/vybrání události. Tato struktura by měla sloužit jako mezičlánek mezi rozhraním vystaveným pro vkládání/vybírání události a technologií, která teprve provede samotné vložení/vybrání serializované události, včetně metadat. Atributy struktury, poté vycházely ze samotného případu užití a sice serializovaná událost do námi zvoleného formátu, název třídy události včetně jmenného prostoru, jenž je událost instancí a do níž se má deserializovat, context, aplikace ve které byla událost vyvolána, poté nepovinný atribut názvu kompresního algoritmu, pokud byl použit a datový formát do jehož je daná událost serializována. Pro podporu odpovědi na událost bylo nutné ještě přidat do struktury nepovinný atribut představující řetězec pro unikátní párování událostí a jejich odpovědí, přítomný pouze v případě, že je očekávána odpověď. Dále rozšíření o hodnotu výčtového typu zda se jedná o událost, nebo odpověď. V PHP je tato struktura vyjádřena jako rozhraní obsahující metody pro získání výše zmíněných atributů. Jelikož je pro událost i odpověď struktura totožná, nese jméno podle společného předka těchto tříd, tedy `APayload` a je rozeznávána řetězcem, neboť PHP jako takové výčtové typy nepodporuje a sice buďto se jedná o událost nebo o odpověď.

```php
interface IPayloadMetadata {

   public function getContext(): string;

   public function getContentType(): string;

   public function getCompressionAlgorithm(): ?string;

   public function getTargetClass(): string

   public function getSerializedPayload(): string;

   public function getResponseCode(): ?string;

   public function getType(): string;
}
```

Výpis 9: Rozhraní IPayloadMetadata

Důvodem proč je zvoleno rozhraní a není implementována přímo třída, je ten, že je žádoucí nechat možnost na konkrétní implementaci technologie fronty a vystavit pouze rozhraní přístupu. Konkrétní implementace poté může pouze obalit potřebná data do tohoto rozhraní a tím zjednodušit implementaci a předejít zbytečnému kopírování hodnot. Nicméně byla vytvořena i třída implementující toto rozhraní, vycházející z datového typu mapa a je použita jako výchozí implementace v neveřejné metodě *toPayloadMetadata*, která provádí konverzi `APayload`, předek události/odpovědi, do rozhraní `IPayloadMetadata`. Konverze je standardně prováděna z předaných parametrů `APayload` a nepovinného parametru `EventResponseLink`. Atribut *targetClass* je název instance třídy dědící z `APayload`, včetně prostoru jmen, *serializedPayload* je do formátu JSON serializovaný argument `APayload`, *responseCode* je řetězec získaný z `EventResponseLink`, a to pouze v případě že je parametr předán, jinak null. Atribut *contentType*, je pevně nastavený podle serializace na datový formát JSON, atribut komprese, která se ve výchozím nastavení nepoužívá je vždy null a atribut *context*, jenž je název aktuální aplikace se získává z aktuální instance `EventQueue`. K metodě *toPayloadMetadata* je implementována i její zrcadlená metoda *fromPayloadMetadata*, která provádí opak a sice konverzi rozhraní `IPayloadMetadata` do `APayload`. Obě tyto metody jsou přetížitelné v potomku a zaručující tedy výše zmíněnou flexibilitu pro implementace s konkrétní technologií.

Všechny předcházející kroky nutné ke zdárné implementaci metod z rozhraní `IEventQueue` jsou tedy splněny a je možno přejít k implementaci metod samotných. Jako první metoda *enqueue* přijímající parametr typu `AEvent` s návratovou hodnotou void, tento parametr převede pomocí metody *toPayloadMetadata*, volané pouze s parametrem `AEvent`, do formátu implementující rozhraní *IPayloadMetadata* a dále jej deleguje do abstraktní metody *_enqueue*, která přijímá právě parametr typu `IPayloadMetadata` s návratovým typem void a je implementována až ve frontě využívající konkrétní technologii. Jako druhá metoda *dequeue*, nepřijímá žádný parametr a vrací hodnotu typu `AEvent` nebo null. V této metodě jako první voláme abstraktní metodu *_dequeue*, která je implementována až ve frontě využívající konkrétní technologii, s návratovou hodnotou `IPayloadMetadata` nebo null. V případě, že metoda vrátí null je tato hodnota delegována, tudíž taktéž metoda *dequeue* vrací null. V případě, že ale návratová hodnota je typu `IPayloadMetadata`, tak je tato hodnota předána výše popsané metodě *fromPayloadMetadata* pro konverzi, poté dále přetypována na `AEvent` a nakonec vrácena. Nyní zbývala implementace rozšíření v podobě rozhraní `IEventQueueQueryable`, `IEventQueueConversable` a `IEventQueueRespondable`. Všechna tato rozšíření jsou v PHP implementována pomocí konstrukce jazyka známé jako trait, jenž umožňuje jednodušší znovupoužití kódu mezi vybranými třídami, přímým vložením kódu do třídy.

První trait je `TEventQueueQueryable` implementující rozhraní `IEventQueueQueryable` a tudíž jedinou metodu *queryQueue* přijímající jako parametr řetězec, reprezentující prostor jmen,

ve kterém se událost nachází s návratovou hodnotou `AEvent` nebo null. V této metodě jako první voláme abstraktní metodu *_queryQueue*, implementovanou až v třídě využívající konkrétní technologii fronty, přijímající námi delegovaný řetězec reprezentující prostor jmen s návratovou hodnotou `IPayloadMetadata` nebo null. V případě, že metoda vrátí null je tato hodnota delegována, tudíž taktéž metoda *queryQueue* vrací null. V případě že ale návratová hodnota je typu `IPayloadMetadata`, tak je tato hodnota předána *fromPayloadMetadata* pro konverzi, poté dále přetypována na `AEvent` a nakonec vrácena.

Další takovouto vytvořenou trait je `TEventQueueConversable` implementující rozhraní typu `IEventQueueConversable`. Nutné je implementovat dvě metody z toho první metoda *request* přijímající parametr `AEventRespondable` s návratovou hodnotou typu `EventResponseLink`. Parametr `AEventRespondable`, spolu s jeho atributem `EventResponseLink` je předán metodě *toPayloadMetadata* pro konverzi do `IPayloadMetadata` a následné delegaci hodnoty abstraktní metodě *_request*, která je implementována až ve třídě implementující konkrétní technologii fronty. Metoda nakonec vrací atribut `EventResponseLink` z parametru `AEventRespondable`. Druhou metodou v rozhraní je *retrieveResponse* přijimající parametr `EventResponseLink` s návratovou hodnotou `AEventResponse` nebo null. Jako první opět delegujeme parametr abstraktní metodě *_retrieveResponse*, implementované až v třídě s konkrétní technologií fronty, která vrací návratovou hodnotou `IPayloadMetadata` nebo null. V případě, že metoda vrátí null je tato hodnota delegována, tudíž taktéž metoda *retrieveResponse* vrací null. V případě že ale návratová hodnota je typu `IPayloadMetadata`, tak je tato hodnota předána *fromPayloadMetadata* pro konverzi, poté dále přetypována na `AEventResponse` a nakonec vrácena.

Poslední vytvořenou trait je `TEventQueueRespondable` implementující jediné rozhraní typu `IEventQueueRespondable` a tudíž jedinou metodu *respond* přijímajicí dva parametry první typu `AEventResponse` a druhý `EventResponseLink` s návratovou hodnotou void. V této metodě jako první delegujeme přijaté parametry metodě *toPayloadMetadata* a její vrácenou hodnotu zase předáme abstraktní metodě *_respond*, implementovanou až v třídě s konkrétní technologií fronty, přijímající `IPayloadMetadata` s návratovou hodnotou void.

## 4.3 Úkol třetí – Implementace Event Queue s MySQL

### 4.3.1 Zadání úkolu

Druhým úkolem bylo vytvořit Event Queue tentokrát však již implementující konkrétní technologii fronty a sice MySQL [6]. Toto obnáší navrhnutí databázového schématu, včetně implementace metod splňující rozhraní Event Queue a všech jeho rozšíření.

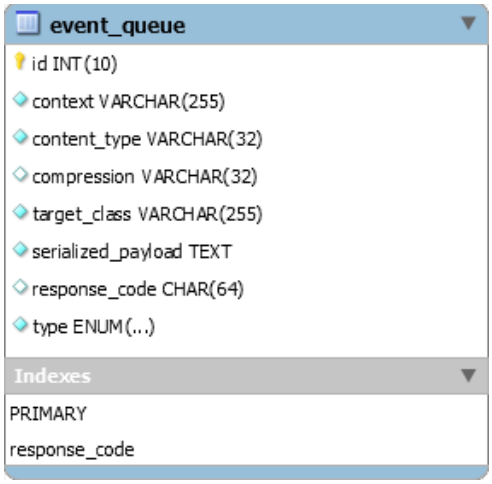
### 4.3.2 Zadání úkolu

Jak již bylo zmíněno v zadání, bylo nutné navrhnout databázové schéma, nutné k uložení hodnot struktur definovaných při implementaci Event Queue rozhraní. Dále vytvořit balíček MySQL

Event Queue, který bude rozšiřovat balíček Event Queue a konkretizovat abstraktní metody, které balíček vystavil, čímž bude docíleno funkcionalit závislých na konkrétní technologii.

### 4.3.3 Implementace

Návrh databázového schématu se zprvu zdál obtížnější než ve skutečnosti byl. Velké zjednodušení přineslo unifikované rozhraní `IPayloadMetadata`, pro všechny funkcionality vystavené Event Queue, díky tomuto rozhraní stačilo navrhnout jedinou tabulku, do které se rozhraní uloží a nad kterou se provádějí veškeré dotazy. Výsledný návrh tedy obsahuje sloupce přímo podle datových typů atributů rozhraní `IPayloadMetadata` s výjimkou atributu *type*, protože MySQL podporuje výčtový typ a není tedy potřeba jej nahrazovat řetězcem jak je tomu v PHP. Sloupec *response_code* je poté ještě indexován, pro lepší výkon při párování událostí a odpovědí. Tabulka obsahuje ještě sloupec *id*, jenž je primárním klíčem a díky této jedinečnosti slouží k usnadnění mazání a aktualizace jednoho řádku z tabulky.




Obrázek 2: Event Queue tabulka

Po návrhu databázového schématu, jsem mohl přejít k vytvoření balíčku MySQL Event Queue, který staví na balíčku Event Queue a konkretizuje metody vystavené tímto balíčkem a dále je závislý na neveřejném balíčku Connetions, dostupném pouze pro firemní projekty, který se stará o spojení a vystavění rozhraní pro komunikaci s databází. Poslední závislostí je balíček PHPUnit [3], který slouží jako základ pro psaní jednotkových testů a nabízí rozhraní pro tvorbu mocků, jenž slouží jako náhražka za reálně implementace. V balíčku MySQL Event Queue jsem započal implementaci vytvořením třídy `MySQLEventQueue` dědící z abstraktní třídy `AEventQueue` implementovanou v balíčku Event Queue. Protože `MySQLEventQueue` už není abstraktní, ale finální implementací musím tedy implementovat všechny abstraktní metody vystavené rodičem, které bez rozšíření jsou *__enqueue* a *__dequeue*. Dále jsem vytvořil třídu `MySQLEventQueueTest`, dědící z `TestCase` a napsal testovací scénář pro obě veřejné metody *enqueue* a *dequeue*. Scénář jsem se snažil držet jednoduchý a přímočarý, tudíž obsahoval pouze vložení tří událostí do

fronty metodou *enqueue* a jejich následné vybrání z fronty metodou *dequeue*. Než jsem se pustil do implementace metod, pečlivě jsem zvážil jaké funkce vlastně budu pro všechny metody, včetně rozšíření implementovat. Došel jsem k závěru, že bude potřeba implementovat pouze dvě neveřejné metody pro vložení záznamu a pro jeho výběr. Metodu pro vložení jsem nazval *insert* a přijímala pouze parametr `IPayloadMetadata` s náratovým typem void. Metodu pro výběr zase *select* ta přijímá nepovinný parametr typu `Where`, která je implementován v balíčku Connections a jak už název napovídá, jedná se o strukturu popisující SQL klauzuli where, které slouží k filtraci dotazu. Metoda měla i návratový typ `IPayloadMetadata` nebo null. Zbývalo tyto metody jen implementovat, k tomu je potřeba databázové spojení, jenž je zajištěno rozhraním `ConnectionInterface`, definováno v balíčku Connections, které jak radí metodika OOP zvaná vkládání závislostí, vyžadována parametrem konstruktoru třídy `MySQLEventQueue` a poté vložena do neveřejného atributu *connection*. Jako první jsem tedy začal s implementací vkládání, tedy metodou *insert*, která transformuje přijímaný parametr `IPayloadMetadata` do pole klíčů a hodnot, kde klíčem je sloupec v tabulce *event_queue* a takto jej vloží do tabulky. Implementace byla triviální, hlavně díky rozhraní `ConnectionInterface`, která nabízí třídu `QueryBuilder`, která slouží k sestavení SQL dotazu a jeho spuštění jako prepared statement, které řeší problematiku útoku SQL injection. Metoda pro výběr byla znatelně složitější, ikdyž to tak na první pohled tak nevypadalo. Strategie výběru byla jasná, vybrat vždy právě jeden co nejmladší záznam, který má typ buďto událost, nebo odpověď, rozhodováno podle předaného parametru metody, dále vyhovuje-li podmínce v klauzuli where, také předané jako nepovinný parametr metody. V případě, že byl nalezen záznam, je nutné jej z tabulky odstranit. Takovýto záznam poté obalit do rozhraní `IPayloadMetadata` a vrátit, pokud žádný záznam nebyl nalezen, vrátit null. Po implementaci metody jsem se vrátil zpátky k testům, kde zbývalo nakonfigurovat falešnou testovací databázi, která se použije v testech namísto produkční. Po krátké konfiguraci jsem spustil testy a ty zdárně proběhly, pocit nadšení z dobře odvedené práce a neopomnění žádného detailu se záhy vytratil, když jsem byl svým mentorem upozorněn, že výsledná aplikace bude běžet ve více instancích a bude docházet k souběhu, kdy jednotlivé instance budou vybírat stejný záznam, protože se nestihl včas odstranit. Bylo tedy nutné obalit výběr do transakce a vybraný záznam uzamknout pro aktualizaci.

Hotov se základní implementací `EventQueue`, dalším krokem bylo pustit se do implementace rozšíření. Začal jsem s rozšířením `IEventQueueQueryable`, která přidává metodu *queryQueue*, která je implementována v trait `TEventQueueQueryable`, jenž vystavuje abstraktní metodu *_queryQueue*. Přidal jsem testovací scénář a sice když je do fronty vloženo pět událostí se čtyřmi unikátními prostory jmen. Očekávané chování je, pokud se třikrát zavolá metoda *queryQueue* s prostorem jmen, který je unikátní pouze pro dvě z vložených událostí, dvě volání vrátí událost z daného prostoru jmen a třetí vrátí null. Pustil jsem se tedy do implementace *_queryQueue* přijímající parametr typu řetězec, reprezentující prostor jmen, ve kterém se má událost nacházet a návratovou hodnotou `IPayloadMetadata` nebo null. Využil jsem již implementovanou metodu *select*, ale tentokrát bylo nutné předat i nepovinný parametr typu `Where`, který zaručí filtrování.

`Where` třída se dá zkonstruovat ze tří parametrů a sice sloupce, operátoru a hodnoty. V tomto konkrétním případě je to sloupec *target_class*, operátor LIKE a hodnota předaného řetezce s postfixem %, jenž v MySQL reprezentuje žolíkový znak, značící že může následovat libovolný počet znaků, včetně žádného. Po dokončení implementace jsem pustil test s předem popsaným scénářem a jako předtím test prošel a já se mohl pustit do implementace posledního rozšíření v podobě dvou rozhraní `IEventQueueConversable` a `IEventQueueRespondable` jenž přidávají tři metody, které jsou implementovány v trait `TEventQueueConversable` a `TEventQueueRespondable`, kde jsou vystaveny tři abstraktní metody *_request*, *_retrieveResponse* a *_respond*. Opět jsem přidal testovací scénář, který pomocí metody *request* vložil do fronty událost, na který bylo rázem odpovězeno voláním metody *respond* a nakonec byla odpověď vyzvednuta metodou *retrieveResponse*. Začal jsem tedy s implementací metody *_request*, která obsahuje pouze delegaci parametru typu `IPayloadMetadata` do volání metody *insert*. Metoda *_respond* byla stejně jednoduchá. Až poslední metoda *_retrieveResponse* vyžadovala přidat do volání metody *select* argument typu `Where`, který definoval filtr pro sloupec *response_code*, který musel být shodný s předávanou hodnotou z `IPayloadMetadata` atributu *responseCode*. Po implementaci jsem pustil testy s předem zmíněným testovacím scénářem, ale tentokrát testy hlásily chybu, protože metoda *retrieveResponse* nevrátila očekávaný `AEventResponse` ale null. Po krátkém procesu debugování jsem zjistil příčinu. Při volání metody *_retrieveResponse* jsem opomenul změnit hodnotu parametru volání metody *select*, která doposud byla ve všech metodách s hodnotou reprezentující událost na hodnotu reprezentující odpověď. Po této změně, už test proběhl podle očekávání.

## 4.4 Úkol čtvrtý – Implementace Event Server

### 4.4.1 Zadání úkolu

Čtvrtým úkolem bylo vytvořit aplikaci Event server, založenou na frameworku Laravel [2]. Aplikace využívá Event Queue, včetně všech rozšíření jen s minimálními závislostmi na konkrétní technologii fronty. Aplikace má za úkol vykonat událost vybranou z Event Queue, případně sestavit odpověď. Dále bude aplikace poskytovat možnost konfigurace časových rozestupů mezi jednotlivými dotazy na Event Queue.

### 4.4.2 Analýza úkolu

Aplikace implementována jako konzolový příkaz, bude přijímat parametry pro konfiguraci časových rozestupů mezi jednotlivými dotazy na Event Queue. Každá událost ke zpracování bude mapována na jednoho i více odběratelů, jenž bude plnit funkci vykonavatele události. Pokud událost vyžaduje odpověď, jeho odběratelům bude předáno patřičné rozhraní. Smyčka, která bude udržovat aplikaci v běhu bude využívat interní balíček Queue Process Control.

### 4.4.3 Implementace

Prvním krokem bylo založení aplikace, tuto akci musel vykonat můj mentor, neboť já sám jsem na ni v systému neměl práva. Během čekání na založení jsem věnoval čas studiu frameworku Laravel, přesněji jak založit vlastní příkaz. Ukázalo se, že proces je jednoduchý a vše co je nutné je založit třídu, která dědí ze třídy `Command` jmenného prostoru frameworku, vyplnit název včetně přijímaných parametrů a implementovat metodu *handle*. Protože aplikace, ještě nebyla připravena, začal jsem návrhem rozhraní, pro odběratele, které podle definice své funkce přijímá událost s návratovým typem void. Konkrétní implementace odběratele potom provádí přetypování na jím očekávanou třídu události.

```
interface ISubscriber {

   public function subscribe(AEvent $event): void;
}
```

Výpis 10: Rozhraní ISubscriber

Dále bylo nutno navrhnout rozhraní, které bude mapovat událost na konkrétní odběratele. Premisa byla taková, že podle třídy události bude rozhodnuto jací odběratelé událost zpracovávají a jejich instance vráceny v poli. V PHP je takovéto rozhraní implementováno jako metoda, která přijímá řetězec, protože PHP nemá datový typ reprezentující třídu, jako tomu je například v jazyku Java a standardně se tedy používá řetězec a metoda vrací pole odběratelů, bohužel v PHP se nedá pole omezit datovým typem.

```
interface ISubscriberFactory {

   public function mapToSubscribers(string $eventClass): array;
}
```

Výpis 11: Rozhraní ISubscriberFactory

Mezitím už byla aplikace vytvořená a připravena na vývoj a já se tedy pustil do vývoje abstraktních třídy `ARespondableSubscriber` implementující rozhraní `ISubscriber`, která bude sloužit jako základ pro odebírání událostí, které vyžadují odpověď. Třída v konstruktoru vynucuje předání rozhraní `IEventQueueRespondable` a následně jej vkládá do atributu *responser*, který je poté dostupný v potomku a dovoluje tedy odpovědět na událost v libovolné části svého vykonávání. Další třídou, která vyžadovala implementaci je `SubscriberFactory` implementující rozhraní `ISubscriberFactory` a tudíž jedinou metodu *mapToSubscribers*. Pro třídu byl napsán i test `SubscriberFactoryTest` s testovacím scénářem, který očekává, že pro danou událost se vrátí právě dvě instance odběratelů. K docílení této funkcionality `SubscriberFactory` ve svém konstruktoru vyžaduje dva parametry mapu tříd událostí na pole tříd odběratelů a rozhraní `IEventQueueRespondable`, oba parametry jsou uloženy do atributů a používají se v metodě

*mapToSubscribers*, kde předaná třída události se mapuje na pole tříd odběratelů. Poté se pro každou vybranou třídu odběratele vytvoří instance, ale pouze jednou a v případě, že se jedná o třídu s předkem `ARespondableSubscriber`, je mu při instanciaci předán `IEventQueueRespondable`. Takto namapované instance se poté vrátí. Pro spuštění testu bylo nutné vytvořit takovouto mapu, kde se mapovaly mock-up objekty těchto tříd, po dokončení konfigurace test proběhl bez sebemenších problémů. Zbývalo tedy vytvořit třídu `EventListener` dědící z třídy `Command`, prostoru jmen frameworku, implementující název příkazu, který byl zvolen event-listen s parametry podporující nastavení čekaní mezi jednotlivými dotazy na Event Queue, dále parametr pro podporu dotazování se na specifický prostor jmen. Dále je nutné implementovat metodu *handle*, která se volá při spuštění našeho příkazu. Na začátku metody probíhá hromadná inicializace `MySQLEventQueue`, mapy událostí na pole odběratelů, `SubscriberFactory` a `QueueProcessController`, třídy z neveřejného balíčku Queue Process Control, která slouží pro řízení a běhu smyčky na základě definovaných pravidel. Tato pravidla jsou sestavena z parametru pro dobu čekání mezi jednotlivými dotazy na Event Queue. Parametr pro dotazovaní se na specifický prostor jmen naopak slouží k vybrání metody dotazování se na Event Queue buďto *dequeue* v případě, že parametr není vyplněn, nebo *queryQueue* s hodnotou parametru. Třída `QueueProcessController` se poté stará o udržování nekonečné smyčky, která začíná výběrem události z Event Queue, poté mapováním události třídy implementující `ISubscriberFactory` a následné vykonání události vybranými odběrateli. V případě, že nebyla nalezena žádná událost, tak `QueueProcessController` čeká, aby zbytečně nezatěžoval Event Queue.

## 4.5 Úkol pátý – Implementace Event Client a Translation Event

### 4.5.1 Zadání úkolu

Pátým úkolem bylo vytvořit Event Client, který bude sloužit k maskování konkrétní Event Queue technologie. Bude schopen čekat na odpověď a zjednodušovat práci s Event Queue. Dále bude implementován v ukázkové události, která nahradí dosavadní mechaniku reportování nepřeložených textů nazvaná Translation Event.

### 4.5.2 Analýza úkolu

Podle zadání bylo nutné vytvořit balíček Event Client, závislý pouze na Event Queue a nikoliv na konkrétní technologii fronty. Dále implementovat funkcionalitu čekání na odpověď. Hlavní myšlenkou bylo vytvořit abstrakci s možností řady rozšíření, kterou až implementace v konkrétní aplikaci pouze poskládá podle svých potřeb. První použití systému událostí bude při nahrazení dosavadní mechaniky v překladovém sytému.

### 4.5.3 Implementace

Během vytváření balíčku Event Client, jsem přemýšlel o návrhu, který vzešel z analýzy a jak jej co nejlépe realizovat. Po zvážení různých možností implementace, jsem se rozhodl pro vystavení funkcionalit Event Client pomocí dvou trait, které třída v aplikaci bude používat, podle vlastností které vyžaduje a na základě nich vybere konkrétní Event Queue implementaci. První trait `TEventQueue`, která vystavuje metodu *enqueue* z rozhraní `IEventQueue` a nic víc. Pro další `TConversable`, bylo nutné vytvořit třídu `EventPromise`, která je jednoduchou implementací slibu s veřejnými metodami *then* a *catch* přijímající funkci, která se má zavolat při dokončení slibu a metodou *await*, která přijímá parametr typu float, který určuje jakou maximální dobu v milisekundách má metoda na odpověď čekat. Samotná trait `TConversable`, vnitřně používá rozhraní `IEventQueueConversable` pro zasílání a vyzvedávání událostí z Event Queue, ale veřejně vystavuje pouze metodu *dispatch*, která přijímá parametr `AEventRespondable` a vrací právě datový typ `EventPromise`. Dále trait implementuje ještě veřejnou metodu *generateEventResponeLink* pro usnadnění vytváření třídy `EventResponseLink`. Metoda přijímá řetězec reprezentující třídu odpovědi a vrací sestavenou třídu `EventResponseLink`. Třída ve výchozí implementaci je sestavena z přijatého parametru a hodnotou pseudo-náhodného řetězce o délce 32 bytů generovaného pomocí nativní funkce v PHP *random_bytes*. Metoda je přetížitelná pro účely cíleného generování vlastních hodnot *responseCode*, za účelem jako je například možné kešování odpovědí. Pro pohodlnější používání balíčku byla přidána ještě třída `CommonEventClient` používající obě balíčkem definované traity `TEventQueue` a `TConversable` s implementací, která volaní metod *enqueue* a *dispatch* deleguje na metody atributu *queue*, implementující rozhraní `IEventQueue` a `IEventQueueConversable`, předávaného v konstruktoru třídy.

Následovalo použití Event Client v překladovém systému, kde má nahradit dosavadní mechaniku reportování nepřeložených textů. Po důkladném prostudování stávající mechaniky byla vytvořena třída `TranslationEvent` dědící ze třídy `AEvent` a definující atributy jako *language*, jazyk do kterého se měl překlad přeložit, *btf*, řetězec jenž slouží jako kód podle kterého se spolu s jazykem páruje překlad, *withinApplicationContext*, jenž je řetězec pomáhající identifikovat v jakém místě aplikace se překlad nachází, kterého. Třída dále implementuje nativní rozhraní `JsonSerializable`, které jak už název napovídá nutní k implementaci metody *jsonSerialize*, která vrací mapu hodnot pro serializaci do formátu JSON. Poté stačilo vybrat technologii Event Queue, která zatím byla implementována pouze pro MySQL a předat tuto instanci třídě `CommonEventClient`, kde provolat metodu *enqueue* s parametrem `TranslationEvent` a implementace na straně překladového systému je hotová. Zbývá tedy implementace na straně Event Serveru, kde je nutné vytvořit odběratele, tedy třídu `TranslationEventSubscriber` implementující rozhraní `ISubscriber` a tedy metodu *subscribe*, kde se vykoná rutina pro událost typu `TranslationEvent`, která byla převzatá z původní mechaniky a přenesena na Event Server. Dále rozšířit mechaniku o založení úkolu v aplikaci JIRA, která je přístupná přes API, implementované v neveřejném balíčku JIRA API. Rozšíření bylo jednoduché, stačila pouze in-

stanciace třídy `JiraApiClient` a provolání metody *createIssue*, jenž přijímá řetězce reprezentující název a popis vytvářeného problému. Tyto řetězce byly sestaveny z hodnot přítomných v `TranslationEvent` a zbývalo ještě předat jméno tabule, kde má být úkol vytvořen, to bylo pevně stanoveno na hodnotu translations. Před dokončením zbývalo namapovat nově vytvořeného odběratele, `TranslationEventSubscriber` na událost `TranslationEvent` a vyzkoušet implementaci. Po testování a doladění textů a drobných chyb, byla funkcionalita připravena na ostré nasazení na produkčním prostředí.

## 4.6 Úkol šestý – Implementace Event Queue s RabbitMQ

### 4.6.1 Zadání úkolu

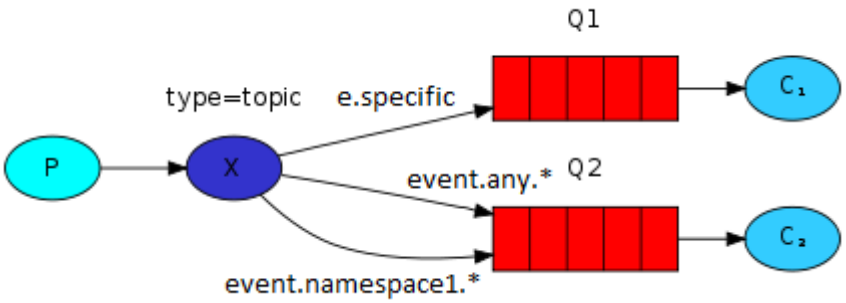
Šestým úkolem byla implementace Event Queue s technologií RabbitMQ [5]. Navrhnout strategii pro směrování zpráv a vybírání front, včetně implementace rozhraní Event Queue a všech jeho rozšíření.

### 4.6.2 Analýza úkolu

Nejprve bylo nutné nastudovat si technologii, poté si nastudovat dokumentaci balíčku php-amqplib, který je použit pro komunikaci s RabbitMQ serverem a dále se podrobněji seznámit s technologií. Poté vytvořit balíček RabbitMQ Event Queue, který bude rozšiřovat balíček Event Queue a konkretizovat abstraktní metody, který balíček vystavil, čímž bude docíleno funkcionalit závislých na konkrétní technologii.

### 4.6.3 Implementace

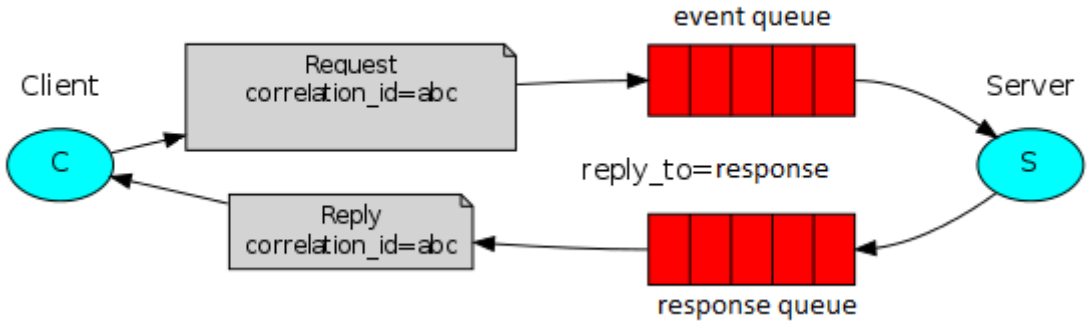
Po seznámení s technologií a její instalaci na vývojářském stroji, jsem mohl začít s experimentováním. Nejprve jsem si prošel tutoriály dostupné v oficiální dokumentaci a zkusil si implementaci pár POC návrhů. Poté jsem se pustil do založení balíčku a návrhu strategie směrování front pro každou z požadovaných funkcionalit. Pro správné pochopení strategie je nutné vysvětlit základní premisu rozesílání událostí.




Obrázek 3: RabbitMQ [5] ukázka strategie

- P je Producer, neboli klient, který produkuje událost
- X je Exchange, neboli výměník, rozesílá událost do front
- Q je Queue, neboli fronta, jsou fronty samotné
- C je Consumer, neboli Event Server, zpracovávající událost

Po dlouhém uvažování a zkoušení byla zvolena strategie, která používá exchange typu topic. To dovoluje v směrovat na základě řetězce, který používá žolíkové značky * pro právě jeden výraz, nebo # pro žádný nebo více výrazů. V implementaci je řetězec odvozen z jmenného prostoru události. Fronty jsou nepersistentní, anonymní a podmíněně vznikají i zanikají spolu s instancí Event Serveru. Pro funkcionalitu odpovědi na událost je vytvořena dedikovaná persistentní fronta s názvem response, do které se posílají všechny odpovědi na událost a jsou párovány podle *correlation_id*, jenž má hodnotu podle atributu *response_code*. *Correlation_id* je součástí AMQP protokolu od verze 0-9-1 a používá se právě pro párování.




Obrázek 4: RabbitMQ [5] odpověď na událost

Po definování strategie, mohla začít samotná implementace třídy `RabbitMQEventQueue`, která dědí z abstraktní třídy `AEventQueue` a implementuje tedy abstraktní metody *_enqueue* *_dequeue*. V konstruktoru třídy je předána instance třídy `AbstractConnection` z balíčku php-amqplib, jenž zprostředkovává spojení s RabbitMQ serverem a slouží k vytváření komunikačních kanálů, ke kterým se váže jednotlivé fronty a exchange. První metodou k implementaci je neveřejná metoda *send*, která přijímá parametr typu `IPayloadMetadata`, který obalí do třídy `AMQPMessage` a posílá na exhange. Druhá neveřejná metoda *receive*, která přijímá nepovinný argument typu řetězec, který slouží k filtrování jmenného prostoru události a je implementována pomocí metody *get*, která vrací právě jedenu AMQP zprávu z již otevřeného a svázaného kanálu, která je transformována zpátky do `IPayloadMetadata` rozhraní a vrácena. Nyní jsem mohl přejít k implementaci metod vystavených rodičem a rozšířeními. Metody *_enqueue* a *_dequeue* pouze delegují volání na metody *send* a *receive*. První rozšíření v podobě rozhraní `IEventQueueQueryable`, které je z části implementováno v `TEventQueueQueryable` a nutí implementovat metodu *_queryQueue*, která přijímá řetězec reprezentující jmenný prostor v němž se

má událost nacházet a toto volání je delegováno i s parametrem na metodu *receive*. Další rozhraní `IEventQueueConversable` spolu s trait `TEventQueueConversable` a `IEventQueueRespondable` s trait `TEventQueueRespondable` vyžadují implementaci metod *__request* která pouze deleguje volání včetně parametru `IPayloadMetadata` na metodu *send*, metoda *__respond* která má velice podobnou implementaci jako metoda *send* s jediným rozdílem, a sice že používá frontu response. Dále metoda *__retrieveResponse* měla opět velice podobnou implementaci jako metoda *receive* s tím rozdílem, že používá frontu response.

Bohužel pro tento balíček nebyly implementovány žádné testy, neboť udělat mock-up objekt abstraktní třídy `AbstractConnection` z balíčku amqp-php nebylo v rozumném čase proveditelné a balíček neposkytuje žádné rozhraní, které by se dalo místo toho využít. Stejně tak nahradit balíčkem vlastním řešením, které by splňovalo funkční požadavky bylo příliš časově náročné.

## Časový plán

V této části, ve stručné tabulce se nachází popis úloh a jejich odhadovaná časová náročnost.

| Úloha | Počet dní |
|---|---|
| Seznámení se s firmou | 1 |
| Příprava pracovního prostředí | 2 |
| Seznámení se s ekosystémem aplikací | 2 |
| Analýza a implementace struktur událostí a odpovědí | 3 |
| Analýza a implementace Event Queue | 6 |
| Analýza a implementace MySQL Event Queue | 10 |
| Analýza a implementace Event Server | 5 |
| Analýza a implementace Event Client | 3 |
| Implementace Translation Event | 3 |
| Analýza a implementace RabbitMQ Event Queue | 15 |

# 5 Využité a nově nabyté znalosti

Do odborné praxe jsem nastoupil sebevědomý a odhodlaný využít znalosti, jenž jsem nabyl při studiu. Věděl jsem, že řešení úkolů nebude jednoduché a nemusí být vždy hned správně, byl to ale jen zlomek zkušeností, které jsem se naučil. Hned ze začátku praxe, kdy jsem byl začleněn do vývojářského týmu a obeznámen s praktikami agilního vývoje, jsem ocenil předmět softwarové inženýrství a zjistil, že i znalosti, kterým jsem předtím nepřikládal velkou váhu jsou v praxi neocenitelné, protože pomáhají efektivně fungovat celému týmu.

První test znalostí, přišel záhy kdy jsme s kolegy diskutovali návrh jednotlivých částí aplikace, dělení závislostí a funkcionalit. Kdy díky zkušenostem z předmětu Vývoj Informačních Systémů, jsem byl schopen do diskuze aktivně přispívat, tak i věcnými argumenty podpořit své návrhy na řešení dané problematiky. Při návrhu databázové struktury jsem využil znalosti nabyté z předmětů zaměřených na databázové systémy. Tyto znalosti jsem využil i při následném dotazování se databáze, kdy bylo důležité korektně navrhnout transakce, tak aby ve skriptech, které dotazování prováděly nedocházelo k nechtěnému souběhu. Dále jsem využil znalosti OOP, které jsem nabyl v rámci studia v nespočtu předmětů zaměřených na programování. Také jsem si osvojil praktiky a zásady TDD, o kterých jsem měl do té doby jen teoretické znalosti, však v praxi jsem poznal jejich reálné výhody, nejen při vývoji, ale i podpoře a odhalování chyb aplikace. Co pro mě bylo novinkou byla práce v týmu. Nikdy předtím jsem ve vývojářském týmu nepracoval a tato zkušenost mi proto dala spousty cenných informací, ať už se jedná o agilní práci týmu, nebo pouhou komunikaci s kolegou ohledně problému, vše je podstatné ve světě softwarového vývoje.

V neposlední řadě jsem si osvojil i doposud mnou nepoznané technologie a postupy pro usnadnění a vývoj software. Jedním z takových nástrojů je Git, distribuovaný systém správy verzí, který jsem při vývoji hojně využíval a přijal za svůj i způsob myšlení, které sebou verzování přináší. Také bych rád zmínil proces posuzování kódu, kterému jsem z počátku přisuzoval jiný smysl a důležitost, než jakou se nakonec ukázalo že má.

# 6 Závěr

Závěrem bych rád vyzdvihl znalosti získané během studia, které jsem využil při praxi a které mi byly tolik nápomocné při řešení úkolů, jež mi byly zadány. Jedná se také o zkušenosti se získáváním nových znalostí, které jsem si za dobu studia osvojil, což značně ulehčilo a zefektivnilo analýzu úkolů, které jsem v praxi vykonával. Také jsem byl rád, když jsem do odborné diskuse mohl přispívat svými argumenty, jejichž problematiku jsem během studia už řešil a mohl tedy nabídnout teoretická řešení problému, nebo vylepšit, či poukázat na nedostatky v návrhu. Zároveň pro mě bylo velkou zkušeností styl práce a programování, kdy vývoj pomocí TDD a následném testování aplikace na dedikovaném testovacím a před-produkčním prostředí, byly věci pro mne zcela nové, jakožto i technologie se kterými jsem se při tomto setkal. Novinkou pro mě bylo i samotné produkční prostředí, kdy aplikace musela být připravena na běh v cloudu v mnoha instancích. Tudíž aplikace musela rozumně přistupovat k prostředkům, které využívala, tak aby nedocházelo k jejich vyčerpání jedinou běžící instancí. V této problematice jsem se nejvíce naučil od svých kolegů, kteří měli s takovýmto vývojem již zkušenosti. Za největší přínos považuji ale zkušenosti získané prací v týmu, protože v takovéto míře jsem se s ní za dobu studia nesetkal. V předmětech jako počítačové sítě, jsme sice spolupráci na projektu s jinými studenty měli, ale byla to spíše kamarádská debata nad problémy, než vážná odborná diskuze. Z mého pohledu byla pro mne praxe byla velice přínosná a to nejen ze studijního a profesního života. Domnívám se, že praxi jsem absolvoval úspěšně, neboť o tom svědčí i skutečnost, že mi bylo nabídnuto, abych ve firmě po ukončení studií zůstal.

# Literatura


[1] THE PHP GROUP. *PHP: Hypertext Preprocessor [online].* © 2001-2019 [cit. 2019-03-24]. Dostupné z: https://php.net

[2] TAYLOR OTWELL. *Laravel - The PHP Framework For Web Artisans [online].* © 2012-2019 [cit. 2019-03-24]. Dostupné z: https://laravel.com

[3] SEBASTIAN BERGMANN. *PHPUnit [online].* © 2019 [cit. 2019-03-24]. Dostupné z: https://phpunit.de

[4] NILS ADERMANN, JORDI BOGGIANO. *Composer - Dependency Manager for PHP [online].*[cit. 2019-03-24]. Dostupné z: https://getcomposer.org/

[5] PIVOTAL SOFTWARE. *RabbitMQ - Messaging that just works [online].* © 2007-2019 [cit. 2019-03-24]. Dostupné z: https://rabbitmq.com

[6] ORACLE CORPORATION. *MySQL - The world's most popular open source database [online].* © 2019 [cit. 2019-03-24]. Dostupné z: https://mysql.com

[7] *DevExtreme dxChart JS knihovna [online].* © 2011-2019 [cit. 2019-03-24]. Dostupné z: https://js.devexpress.com

[8] *The OAuth 2.0 authorization framework [online].*[cit. 2019-03-24].
Dostupné z: https://oauth.net/2/